CCLRC / Rutherford Appleton Laboratory **SC/4.2**
Particle Physics & Astronomy Research Council
Starlink Project
Starlink Cookbook 4.2

Malcolm J. Currie
1998 January 9

C-shell Cookbook

Version 1.1

Abstract

This cookbook describes the fundamentals of writing scripts using the UNIX C shell. It shows how to combine Starlink and private applications with shell commands and constructs to create powerful and time-saving tools for performing repetitive jobs, creating data-processing pipelines, and encapsulating useful recipes. The cookbook aims to give practical and reassuring examples to at least get you started without having to consult a UNIX manual. However, it does not offer a comprehensive description of C-shell syntax to prevent you from being overwhelmed or intimidated. The topics covered are: how to run a script, defining shell variables, prompting, arithmetic and string processing, passing information between Starlink applications, obtaining dataset attributes and FITS header information, processing multiple files and filename modification, command-line arguments and options, and loops. There is also a glossary.

Contents

1 Introduction

Scripting is a powerful and time-saving tool for performing repetitive jobs, creating data-processing pipelines, and encapsulating useful recipes. A script is a text file containing a set of shell commands and constructs that perform a routine task. The former can include UNIX commands like **mv**, **cd**, **awk**, **sed**; other scripts; and private and Starlink applications. You can create and modify scripts with a text editor.

Although UNIX purists recommend the Bourne shell for scripts and the young Turks would advocate the increasingly popular **perl** language, most Starlink commands are C-shell aliases, and so we use that. The C-shell is also easier to learn than the Bourne shell, especially if you are already familiar the C programming language. This cookbook applies to both the C-shell **csh** and its variants like the tc-shell **tcsh**.

This manual illustrates some relevant techniques for creating C-shell scripts that combine S-tarlink software to improve your productivity, without you having to read a UNIX manual. It aims to give practical and reassuring examples to at least get you started. It does *not* offer a comprehensive description of C-shell syntax and facilities.

This is not like other cookbooks in the Starlink series as the possible recipes are limitless. Instead of concentrating on recipes, it therefore focuses on the various ingredients you may need for your own creations. Thus it has a tutorial style more akin to a guide. However, it has been structured with the aim of letting you dip in to find the desired ingredient.

The author welcomes your comments. If you have "How do I do ... in the C-shell" type questions, or suggestions for further recipes and ingredients, please contact the Starlink Software Librarian (`ussc@star.rl.ac.uk`) or the author (`mjc@star.rl.ac.uk`), so that important techniques omitted from this version may be included in future editions.

2 Further Reading

If you want a comprehensive description of C-shell syntax and facilities; check the **man** pages for **csh**. Books exclusively on the C-shell are not as commonplace as you might expect; one such is *Teach Yourself the Unix C shell in 14 days* by David Ennis & James C. Armstrong (SAMS Publishing, Indianapolis, 1994). While there are plenty of UNIX books, they tend to give spartan coverage of the C-shell, often concentrating on the interactive aspects; in many the examples are sparse. One that bucks the trend is *UNIX Shells by Example* by Ellie Quigley (Prentice-Hall, New Jersey, 1997). This has numerous examples, where the function of each line is explained. C-shell is included, and there are chapters on the tools of the trade like **awk** and regular expressions. The chapter entitled *Shell Programming* in *UNIX for VMS Users* by Philip E. Bourne (Digital Press, 1990) is well worth a read, especially for those of you who developed command procedures in Starlink's VMS era, and want to convert them to UNIX scripts. Chapter 49 of *UNIX Power Tools* by Jerry Peek, Tim O'Reilly, & Mike Loukides (O'Reilly & Associates, 1993) has useful summaries and describes some problems with the C-shell.

3 How to use this manual

It is not necessary to read this manual from cover to cover. Indeed some of it will be repetitive if you do. That's deliberate so that once you have the basics, you can then look up a particular ingredient without having to read lots of other parts of the cookbook.

Before you write any scripts you should look at Sections 5 to 8 (except perhaps 5.7).

4 Conventions

`verbatim`	Commands you enter, filenames, parameters values you supply to scripts or tasks appear in teletype fount.
command	The names of UNIX or Starlink commands appear in bold.
term	Special terms explained in the glossary appear in the san serif fount.

Commands to be entered from the terminal are prefixed with a `%` prompt string. You should not type the `%`.

5 Running a script

Unfortunately, we must start with some boring, but important technical stuff. This will tell you how to run your script using the C-shell. The stages are to select the shell, give execute access to the script, and then actually invoke the script using one of three ways.

5.1 Shell selection

The *first line* of your script tells UNIX which shell you want to use. The following selects the C-shell.

```
#!/bin/csh
```

Somewhat perversely, the `#` is actually the comment character. That's UNIX for you. However, its presence alone starting the first line will normally be sufficient to run a script in the C shell.

Most of the examples in this document are script excerpts, and so do not include the `#!/bin/csh`.

5.2 Making a script executable

If you want to run your script by name or with an alias (see below), you must make your script executable like this

```
% chmod +x myscript
```

where `myscript` is your C-shell script. Remember that the `%` is a convention for the shell prompt; you do not type it yourself. You can edit the script without having to make the file executable again.

5.3 Executing a script by name

Once your script has execute privilege, thereafter you can run it like this:

```
% ./myscript
```

or if it is situated in directory `/home/user1/dro/bin` say

```
% /home/user1/dro/bin/myscript
```

would execute your script. This is long-winded if you want to run the the script frequently. To omit the directory path you need to add the current (`.`) or better, the specific directory to your PATH environment variable.

5.4 Using aliases to run a script

The second method is to define an alias. An alias is a shorthand for a long command to save typing. So in fact this method is just a variant of executing a script by name.

```
% alias myscript /home/user1/dro/bin/myscript
```

Thereafter you would only enter `myscript` to run your script. Now this alias only applies to your current process. Even child processes derived from the current process will not inherit the alias. The way to make an alias global, so each new C-shell process recognises the alias, is to define the alias in your `$HOME/.cshrc` file. This file in your login directory defines the 'environment' for each C-shell process.

Here is a simple `.cshrc` file. It sets some global variables: `noclobber` prevents accidently overwriting an existing file and `history` sets the number of previous commands you can recall. It defines three aliases. The first lets you browse through a directory listing. The second reformats the output from **df** on Digital UNIX so that lines do not wrap. The third lists the largest 20 files in a directory. Finally `.cshrc` runs the script `/star/etc/cshrc` to define Starlink aliases (the **source** command is explained in Section 5.6).

```
set noclobber
set history = 50
alias dsd 'ls -l \!* | more'
alias dff "df \!* |awk \
 '{printf("\"%-20.20s%9.8s%9.8s%9.8s%9.8s %s\\n\",'$1,$2,$3,$4,$5,$6)}' "'"
alias big20 'ls -l | sort -k 5 | tail -n 20'
source /star/etc/cshrc
```

5.5 Removing aliases

There are two methods. One is permanent; the other overrides an alias for a single command. To remove an alias permanently use the `unalias` command. This accepts `*?[ ]` wildcards to match the defined aliases.

```
                         # Removes aliases called:
% unalias myscript       # myscript
% unalias kap_*          # kap_ followed by zero or more characters
% unalias compres?       # compres followed by a single character
% unalias [f-hz][2-5]    # One of f, g, h, or z then an integer
                         # between 2 and 5 inclusive
```

To override an alias, precede the alias with backslash. So suppose you have an alias to prevent you accidently removing a file as shown below.

```
% alias rm rm -i
```

In a script where you know that you want to remove a file, you don't want the script to seek confirmation. So suppose you want to delete file `myfile`, then you would have a line like

```
\rm myfile
```

in your script.

5.6 Executing a script in the current process

The final option is to **source** your script so that it runs in the current process. The benefit of this technique is that any aliases defined in the current process will be known to your script. For example,

```
% source myscript
```

runs `myscript`. Any aliases created in `myscript` will be available to the current process, unlike invocation by name. To save typing you can define an alias to source a script. See the next section on package aliases for an example.

5.7 Package aliases

While convenient, the creation of aliases in your `.cshrc` file does have a drawback: if you define many aliases in the `.cshrc` file, it will decelerate process activation. One way around that is to define a few aliases that run other scripts, each of which in turn define many related aliases. Thus you only create the definitions when they are required. This is how most Starlink packages define their commands. Here is an example. Suppose you had a package or a set of related commands called COSMIC installed in directory `/home/user2/dro/cosmic`, you would first place the following line in your `.cshrc` file.

```
alias cosmic 'source /home/user2/dro/cosmic/cosmic.csh'
```

The left quotes mean execute the command between them. So when you enter

```
% cosmic
```

the script `/home/user2/dro/cosmic/cosmic.csh` is run. This file might look like the following.

```
#!/bin/csh
alias abund /home/user2/dro/cosmic/abundance
alias filter 'source /home/user1/abc/bin/garbage.csh'
alias kcorr /home/user2/dro/cosmic/k-correction.csh
alias radial /home/user2/drmoan/noddy/radial
alias seeing $KAPPA_DIR/psf isize=21 psf=psf21
alias zcol kcorr bands=UJR noevol
```

6 Some simple examples

Let's start with a few elementary scripts to show that script writing isn't difficult, and to illustrate some syntax and constructs.

Suppose that we have an editor such as **jed** which retains a copy of the previous version of a file by appending ~ to the filename. If at some time we realise that we really want the former version and to erase the new one we could have a script called `restore` that would perform the operation for an arbitrary file. It might look something like this.

```
#!/bin/csh
rm $1
mv $1~ $1
```

The first line demands that the C-shell be used. The `$1` represents the first argument supplied on the command line. (There is more on this in Section 7.6.) If the command `restore` runs the script, then entering

```
% restore splat.f
```

would remove `splat.f` and replace it with `splat.f~`.

The above script makes some assumptions. First it does not check whether or not the files exist. Let's correct that.

```
#!/bin/csh
if ( -e $1 && -e $1~ ) then
   rm $1
   mv $1~ $1
endif
```

Here we have introduced the **if...then...endif** construct. When the expression inside the parentheses is true, the following statements are executed until there is an **else** or **endif** statement. In this case `-e $1 && -e $1` is true only when both files exist. The `-e` is one of eight file operators for such things as file access and type, and `&&` is a logical AND.

The second assumption in `restore` is that you'll always remember to supply a file name.

```
#!/bin/csh
if ( $#argv == 0 ) then
   echo Error: no file name supplied
else if ( -e $1 && -e $1~ ) then
   rm $1
   mv $1~ $1
endif
```

The script now reports an error message `Error: no file name supplied` if you forget the argument. `$#argv` is the number of arguments supplied on the command line. Also notice the **else if** construct. The `==` tests for equality.

Instead of an error message you could make the script prompt for a filename.

```
#!/bin/csh
if ( $#argv == 0 ) then
   echo -n "The file to restore >"
   set file = $<
else
   set file = $1
endif
if ( -e $file && -e $file~ ) then
   rm $file
   mv $file~ $file
else
   if ( ! -e $file ) then
      echo Error: $file does not exist
   endif
   if ( ! -e $file~ ) then
      echo Error: $file~ does not exist
   endif
endif
```

When the number of arguments is zero, this script issues the prompt `The file to restore >`. The `echo -n` prevents a skip to a new line, and so allows you to enter your response after the prompt string. `set file $<` equates your response to a shell variable called `file`. If on the other hand you gave the filename on the command line `set file = $1` assigns that name to variable `file`. When we come to use the variable in other commands it is prefixed with a dollar, meaning "the value of the variable". Finally, the script now tells you which file or files are missing. Notice how the variable value can be included in the error message.

For your private scripts you need not be as rigorous as this, and in many cases the simplest script suffices. If you intend to use a script frequently and/or sharing with colleagues, it's worth making a little extra effort such as adding commentary. Comments will help *you* remember what a lengthy script does after you've not run it for a while. It might seem tedious at the time, but you will thank yourself when you come to run it after a long interval. Comment lines begin with `#`. There are comment examples in some of the longer scripts later in the cookbook.

7 Shell Variables

The shell lets you define variables to perform calculations, and to pass information between commands and applications. They are either integers or strings, however it is possible to perform floating-point arithmetic by using certain applications in your script (see Section 10.3). You don't need to declare the data type explicitly; the type is determined when you assign a value to a variable.

Variables are only defined in the current process. If you want global variables, *i.e.* ones that are available to all processes, you should assign them in your `.cshrc` file.

Variable names comprise up to 20 letters, digits, and underscores; and should begin with a letter or underscore.

7.1 Assigning scalar values

You assign a value to a variable using the **set** command.

```
set colour = blue
set title = "Chiaroscuro 1997"
set caption = "The most distant quasar observed is at redshift "
set flux_100 = 1.2345E-09
set n = 42
set _90 = -1
```

The first four examples assign strings, and the last two assign integer values. Yes the value of `flux_100` is not a real number. Multi-word strings should be enclosed in " quotes. The spaces around the equals sign are necessary.

You can also remove variables with the **unset** command. This accepts `*?[ ]` wildcards to match the names of the shell variables.

```
unset colour
unset iso_*            # iso_ followed by zero or more characters
unset flux_1??         # flux_1 followed by any pair of characters
unset [nx-z][0-9]*     # One of n, x, y, or z then an integer followed
                       # by zero or more characters
```

7.2 Assigning arrays

The **set** command is again used but the elements are space-separated lists enclosed by parentheses. Somewhat surprisingly for a shell that mimics C, array elements start at 1, like Fortran. Here are some illustrations.

```
set colours = (blue yellow green red pink)
set label = ("Count rate" Frequency "Integration time (sec)")
set prime = (2 3 5 7 11 13 17 19 23)
```

The first element of `colours` is `"blue"`, the second is `"yellow"` and so on. Multi-word elements must be in " quotes. So the first element of `label` is `"Count rate"`, and the second is `"Frequency"`. The seventh element of `prime` is `17`.

7.3 Using the values of variables

To obtain the value of a variable, you prefix the variable's name with the dollar sign.

```
set count = 333
echo Number of runs is $count
```

would write `Number of runs is 333` to standard output.

```
set caption = "The most distant quasar observed is at redshift "
set z = 5.1
echo $caption$z
```

This will echo `The most distant quasar observed is at redshift 5.1`.

```
if ( $n > 10 ) then
   mem2d niter=$n out=deconv
   display pltitl="Result of deconvolution after $n iterations" accept
endif
```

This tests whether the value of variable `n` is greater than ten, and if it is, executes the statement within the **if. . . endif** structure. The value is also passed into an integer parameter of the application **mem2d**; and a string parameter of application **display**, so if `n` were forty, the resultant value would be `"Result of deconvolution after 40 iterations"`.

Arithmetic, logical and string operations are presented in Sections 10 and 11.

The values of array elements are selected by appending the element index in brackets. This would echo to standard output the string `Starting search at co-ordinates x=300, y=256.`

```
set coords = (300 256)
echo "Starting search at co-ordinates x=$coords[1], y=$coords[2]."
```

The following illustrates use of a text array to specify the annotations of a plot created with application **linplot**.

```
set labels = ("Elapsed time" Flux "Light curve of 3C273")
linplot abslab=$labels[1] ordlab=$labels[2] pltitl=$labels[3]
```

There are some shorthands for specifying subsets of an array. The best way to describe them is through some examples. All the values assume `set prime = (2 3 5 7 11 13 17 19 23)`.

Syntax	Meaning	Value
`$#prime`	Number of elements of variable `prime`	`9`
`$prime[*]`	All elements of variable `prime`	`2 3 5 7 11 13 17 19 23`
`$prime[$]`	The last element of variable `prime`	`23`
`$prime[3-5]`	The third to fifth elements of variable `prime`	`5 7 11`
`$prime[8-]`	The eighth to last elements of variable `prime`	`21 23`

Here we bring some of these ingredients together. Suppose we are experimenting trying to find the most suitable reddish background (`palnum=0`) colour for image display using the **palentry** command from KAPPA. The script below loops using a **while...end** construct: all the commands inside the construct are repeated until the condition following the **while** is satisfied. That might seem a little odd here as we appear to repeat the same commands *ad infinitum* because the number of colours is fixed. That doesn't happen because of the C-shell **shift** command. It discards the first element of an array and reduces the indices of the remaining elements by one, so the original `$colours[2]` becomes `$colours[1]` and so on. This means that the chosen background colour is always `$colours[1]`. The **shift** also decrements the number of colours by one. So the script changes the background colour and pauses for five seconds for you to consider the aesthetics.

```
set colours = (red coral hotpink salmon brown sienna tan)
while ( $#colours > 0 )
   echo "Trying $colours[1]"
   palentry palnum=0 colour=$colours[1]
   sleep 5
   shift colours
end
```

7.4 Special characters

The shell has a number of special characters otherwise known as metacharacters. These include wildcard characters such as `*?[ ]`, single and double quotes, parentheses, and the `\` line continuation.

If you assign a value to a variable or application parameter which includes one or more of these metacharacters, you need to switch off the special meanings. This operation is called escaping. Single characters may be escaped with a backslash. For multiple occurrences single quotes `' '` will pass the enclosed text verbatim. Double quotes `" "` do the same as single quotes except that `$`, `\`, and left quote ‘ retain their special meaning.

```
display pltitl=\"Syrtis Major\" \\
set metacharacters = '[]()/&><%$|#`@''"'
stats europa'(200:299,~120)'
stats europa"(200:299,~$y)"
```

In the first example the double quotes are part of parameter PLTITL (needed because of the embedded space) so are escaped individually. On its own `\` means continue a line, but for Starlink tasks it is shorthand for the `accept` keyword. So we have to tell the shell to treat the backslash literally by preceding it with backslash!

In the last pair of examples an NDF section (see SUN/95's chapter called "NDF Sections") is specified. As the last contains a variable value to be substituted, the `$` retains its normal special meaning but the parentheses are escaped by surrounding the section in double quotes.

7.5 Prompting

Some scripts might have parameters that cannot be defaulted, and so if their values are not given on the command line, the script needs to prompt for them.

```
echo -n "Give the smoothing size"
set size = $<
```

This will prompt for the parameter and store the entered value into shell variable `size`.

7.6 Script arguments and other special variables

The C-shell has some special forms of variable. We have seen some already: the `$<` for prompting, and how to specify array elements. The remainder largely concern command-line arguments. Here they are tabulated.

Syntax	Meaning
`${0}`	The name of the script being run
`$?name`	Returns `1` if the variable `name` is defined, or `0` if it is not defined
`$n`	The value of the `n`th argument passed to the script
`$argv[n]`	The value of the `n`th argument passed to the script
`$#argv`	The number of arguments passed to the script
`$*`	All the arguments supplied to the script
`$$`	Process identification number (useful for making temporary files with unique names)

Thus inside a script, any command-line arguments are accessed as shell variables `$1`, `$2` … `$#argv`. There is no practical limit to the number of arguments you can supply.

Let's look at a few examples. Below is a script `argex.csh`.

```
#! /bin/csh
echo ${0}
echo "Number of arguments is $#argv"
echo $2
echo $argv[2-3]
echo $argv[$]
exit
```

Then we run it with four arguments.

```
% argex.csh "hello world" 42 3.14159 "(300:400,~100)"
argex.csh
Number of arguments is 4
42
42 3.14159
(300:400,~100)
```

Note that you must enclose any argument passed to a script that contains a space or shell metacharacter in double quotes (").

You can see an example using `$*` to process a list of files in Section 12.2.2. There are other examples of script arguments in Section 14.

7.7 Predefined variables

The C-shell has some predefined variables. We've already met `argv`. The most useful other ones are listed below.

Shell variable	Meaning
`cwd`	Current working directory
`home`	Home directory
`path`	List of directories in which to search for commands
`status`	The status returned by the last command, where `0` means a successful completion, and a positive integer indicates an error, the higher the value, the higher the severity.
`user`	Username

So if you wanted to include details of the user and directory at the head of some data-processing log you might write this in your script.

```
% echo "Processed by:        $user" > logfile
% echo "Directory processed: $path" >> logfile
```

This writes a heading into file `logfile` saying who ran the script, and then appends details of the directory in which processing occurred. Note the different metacharacters, namely `>` redirects the output to a file, and `>>` appends to an existing file.

8 Executing a Starlink Application

Running Starlink tasks from a script is much the same as running them interactively from the shell prompt. The commands are the same. The difference for shell use is that you should provide values on the command line (directly or indirectly) for parameters for which you would normally be prompted. You may need to rehearse the commands interactively to learn what parameter values are needed. Although there is less typing to use a positional parameter for the expression, it's prudent to give full parameter names in scripts. Positions might change and parameter names are easier to follow. CURSA is an exception. For this package you should list the answers to prompts in a file as described in Section 12.6.

The script must recognise the package commands. The options for enabling this are described below. Then you can run Starlink applications from the C-shell script by just issuing the commands as if you were prompted. You do *not* prefix them with any special character, like the % used throughout this manual.

If you already have the commands defined in your current shell, you can **source** your script so that it runs in that shell, rather than in a child process derived from it. For instance,

```
% source myscript test
```

will run the script called `myscript` with argument `test` using the current shell environment; any package definitions currently defined will be known to your script. This method is only suitable for quick one-off jobs, as it does rely on the definition aliases being present.

The recommended way is to invoke the startup scripts, such as `kappa`, `ccdpack` within the script. The script will take a little longer to run because of these extra scripts, but it will be self-contained. To prevent the package startup message appearing you could temporarily redefine **echo** as shown here.

```
alias echo "echo > /dev/null"
kappa
ccdpack
unalias echo
```

In traditional UNIX style there is a third option: you could add the various directories containing the executables to your PATH environment variable, however this will not pick up the synonym commands.

```
setenv PATH $PATH:/home/user1/dro/bin:/home/user2/drmoan/noddy
```

As most of the examples in this document are script excerpts, and for reasons of brevity, most do not define the package commands explicitly.

8.1 Parameter files and the graphics database

If you run simultaneously more than one shell script executing Starlink applications, or run such a script in the background while you continue an interactive session, you may notice some strange behaviour with parameters. Starlink applications uses files in the directory $ADAM_USER to store parameter values. If you don't tell your script or interactive session where this is located, tasks will use the same directory. To prevent sharing of the parameter files use the following tip.

```
#!/bin/csh
mkdir /user1/dro/vela/junk_$$
setenv ADAM_USER /user1/dro/vela/junk_$$

<main body of the script>

\rm -r /user1/dro/vela/junk_$$
# end of script
```

This creates a temporary directory (`/user1/dro/vela/junk_$$`) and redefines $ADAM_USER to point to it. Both exist only while the script runs. The $$ substitutes the process identification number and so makes a unique name. The backslash in `\rm` overrides any alias `rm`.

If you are executing graphics tasks which use the graphics database, you may also need to redefine $AGI_USER to another directory. Usually, it is satisfactory to equate $AGI_USER to the $ADAM_USER directory.

9 Passing information between Starlink applications

In scripts you will often want to take some result produced or calculated by one application, and to pass that information to another. Two techniques are available: piping or output parameters.

9.1 Parsing text output

If the first task prints the required value, it is possible to use one or more of **grep**, **sed**, and **awk** to locate and isolate the value. For example, to obtain the mean value of a data array you could have the following lines in your script.

```
set dummy = `stats accept | grep "Pixel mean"`
set mean = $dummy[4]
```

The `accept` keyword tells **stats** to use the current values for any parameters for which it would otherwise prompt you. The back quotes (` ` are important. They tell the shell to execute the expression they enclose, in this case `stats accept | grep "Pixel mean"`, and assign the result to variable `dummy`. So the KAPPA **stats** task is run on the current dataset, and the output is passed to **grep** which searches for the line containing the string `Pixel mean`, and the mean value. When you equate a variable to a multi-word string not enclosed in quotes—words being separated by spaces—the variable becomes an array, whose elements are each assigned to a word. So in this case the fourth word or element is the mean value.

Besides being inelegant, this method demands that the format and text of the output be fixed. So it should be avoided where the first task writes results to output parameters.

9.2 Via Parameters

There is a more-robust technique for passing information between Starlink applications; it does not rely on the formatting of the output. Where an application writes output parameters (otherwise called results parameters), you may use the **parget** command of KAPPA to write to standard output the value or values associated with a parameter for a named application. In a script we want to assign the value to a shell variable. Let's see how that compares with obtaining the same mean value as before.

```
stats accept > /dev/null
set average = `parget mean stats`
```

This runs **stats**, but redirects the output to the null device as the output is not wanted. However, this does not prevent the output parameter called MEAN from being assigned the mean value. **parget** retrieves this value, which is then assigned to variable `average`. Note that **parget** retrieves parameter values for the last invocation of the task. Thus if you wanted the standard deviation too, you would only need to issue the **stats** command once (as shown below).

```
stats \\ > /dev/null
histpeak use=a sfact=0 device=! accept > /dev/null
set mean = `parget mean stats`
set stdev = `parget sigma`
set kurtosis = `parget kurt histpeak`
```

The kurtosis is obtained via the KURT output parameter of the **histpeak** command of ESP.

10 Arithmetic

10.1 Integer

You may include integer arithmetic in your scripts. First you must assign a value to each variable used with the **set** command.

```
set count = 1
set data = (10 24 30 17)
set exception = -999
set list
```

Notice that arrays are defined as spaced-separated values in parentheses. So `$data[3]` is 30. You can also set a variable to a null value, such as `list` above, so that the variable is ready to be assigned to an expression.

To perform arithmetic the expression is prefixed with the special `@` character like this.

```
@ count = $count + 1
@ count++
```

Both of these add one to the value of `count`. Note that the space following the `@` and around the `=` and `+` operator are needed. Likewise the examples below both subtract one from the value of `count`.

```
@ count = $count - 1
@ count--
```

There are several other operators, the most important ones are illustrated below.

```
@ ratio = $count / 5
@ octrem = $data[2] % 8
@ numelement = $i * $j
```

The first divides the value of `count` by 5, rounding down, so if `count` was 11, `ratio` would be 2. The second assigns the remainder of the second element of the array called `data` after division by 8. So if `data[2]` is 30, `octrem` would be set to 6. Finally `numelement` is equated to the product of variables `i` and `j`.

The precedence order is `* / %` followed by `+ -`. If you are in doubt, use parentheses to achieve the desired order.

10.2 Logical

As variables are either integers or strings, you have to find an alternative route to have logical variables in your script. One way is to define string values "true" and "false" (or uppercase equivalents) and test for these. For instance, this checks the value of variable smooth; when it is true the current dataset is Gaussian smoothed.

```
if ( $smooth == "true" ) then
   gausmooth \\
end
```

Remember that UNIX is case sensitive, so for instance, TRUE and true are not the same.

Another popular way is to use an integer, which is assigned to zero for a false value, and 1 for true. Indeed this is the way logical expressions are evaluated by the shell. For instance in the following logical expression

```
@ x = ($n < 5 || 20 <= $n)
@ y = ( !($n < 5 || 20 <= $n) )
```

variable x is 0 (false) when n lies between 5 and 19 inclusive, but is 1 (true) otherwise. Variable y is the negation of x. Note the parentheses surrounding the logical expressions; they are needed when an expression contains >, <, |, or &. A list of the comparison operators is given in the table to the right.

Comparison operators	
Operator	Meaning
`==`	Equal to
`!`	Boolean NOT
`!=`	Not equal
`&&`	Boolean AND
`||`	Boolean OR
`>`	Greater than
`<`	Less than
`>=`	Greater or equal to
`<=`	Less than or equal to

Here are a few more examples.

```
@ flag = ( $name != "abc" )
@ integrate = ( $flux[4] > 20 && $band != 5 )
@ finish = ( $finish && ( $count > $number || $flag == 0 ) )
```

The additional parentheses in the expression for finish indicate the evaluation order starting from the innermost parentheses.

The C-shell also offers some operators that test the characteristics of a file, such as file existence. See Section 12.4 for details.

10.3 Floating Point

The shell cannot perform non-integer arithmetic. Thus you need to employ some other utility such as **bc** or the KAPPA's **calc** command. Here we shall just deal with the latter.

The **calc** tool evaluates Fortran-like expressions, sending the result to standard output. So suppose that we wish to add two real values.

```
set a = 5.4321
set b = 1.2345
set c = `calc $a+$b`
```

The variable `c` takes the value of adding variables `a` and `b`. Note the back quotes that cause **calc** to be executed.

This is fine if you know that the values will always be positive. However, **calc** will object if there are adjacent mathematical operators, such as `+-` as would happen if `b` were negative. So surround variables in parentheses, remembering that the `( )` are metacharacters. See Section 7.4 for more details.

Let's redo the first example along with a few more to illustrate the recommended syntax. This time we'll specify the expression by parameter name rather than position.

```
set a = 5.4321
set b = -1.2345
set c = `calc exp="($a)+($b)"`                      # c = 4.1976
set d = `calc exp="'( $a - 0.5 * ($b) ) / 100 '"`  # d = 0.0481485
set e = `calc exp="(($a)+($b))**3"`                 # e = 296.2874
set f = `calc exp="'($a + ($b)) ** 3'"`             # f = e
```

Don't be intimidated by the surfeit of quotes. The `" "` are needed because we want the dollar to retain its variable-substitution meaning. If the expression contains embedded spaces (usually for clarity) it should be enclosed in single quotes as shown in the assignment to variable `f`. So in general the recipe for using **calc** is

`set` *variable* `= `calc exp="'`*expression*`'"``

Don't try to escape all the metacharacters individually because it can soon become very messy, error prone, and hard to follow.

The default precision is single precision. If you need more significant figures then append `prec=_double` to the **calc** command line. The special symbol `pi` has the value of π. So

```
set arc = 0.541209
set darc = `calc exp=$arc*180.0/pi prec=_double`
```

converts an angle 0.541209 radians (stored in `arc`) to degrees using double-precision arithmetic.

It is sometimes easier to assemble the expression in another variable. Here's another recipe which demonstrates this approach. Suppose that you want to find the average median value for a series of NDFs.

```
# Estimates the median of each sky frame and subtracts it from that image

# Enable KAPPA commands.  Initialsie variables.
kappa > /dev/null
set first = 0
set numndf = 0

# Loop for all NDFs
foreach file (*_fl.sdf)

# Obtain the NDF's name.
   set file1 = $file:r

# Assign the median value of the frame.  HISTAT computes the median,
# and PARGET extracts it from HISTAT's MEDIAN output parameter.
# Dispose of the HISTAT text output to the null file.
   histat $file1 \\ > /dev/null
   set median = `parget median histat`

# Subtract the median from the input NDF.
   csub $file1 $median  $file1"_m"

# Count the number of NDFs.
   @ numndf = $numndf + 1

# Form the string to sum the median values using CALC.  The shell
# only permits integer arithmetic.  The expression is different
# for the first file.
   if ( $first == 0 ) then
      set expr = "( "$median
      set first = 1

# Append a plus sign and the next median value.
   else
      set expr = `echo $expr" + "$median`
   endif

end

# Complete the expression for evaluating the average of the frame
# medians.
set expr = `echo $expr" ) / " $numndf`

# Evaluate the expression and report the result.
echo "Average median is "`calc $expr`
```

The script builds an expression to evaluate the average of the median values. So the first time it has to write the left parenthesis and first median value into string variable `expr`, but otherwise each new value is appended to the expression. Once all the medians are evaluated, the remainder of the expression, including division by the number of NDFs is appended. Finally, the expression is evaluated using **calc**. If you want to learn more about the `set median = `parget median histat`` command see Section 9.2.

10.4 Intrinsic Functions

If you want to include intrinsic functions, such as logarithms and trigonometric functions in your calculations, or perhaps you need some function for an integer expression that is not supplied by the C-shell, such as finding the maximum or the absolute value, **calc** may be the solution. It offers the 28 functions tabulated below.

Here are a few examples.

```
set a = 5
set b = (3 4 6 -2)
set c = `calc exp="'min( $a, $b[1], $b[2], $b[3] )'"`
set d = `calc exp="'dim( $b[4], $a ) + dim( $b[3], $a )'"`
```

The first expression finds the minimum of the four integers, namely `3` and assigns it to variable `c`. The second expression sums two positive differences: `0` and `1` respectively.

```
set mag = `calc exp="'-2.5 * LOG10( $counts )'"`
set separation = `calc exp="atand2(35.3,$dist)"`
echo The nearest index is `calc exp="nint($x+0.5)"`
```

Variable `mag` is the magnitude derived from the flux stored in variable `counts`. `separation` is assigned the inverse tangent of 35.3 divided by the value of variable `dist` measured between −180 and 180 degrees.

Function	Number of arguments	Description
SQRT	1	square root: $\sqrt{\mathrm{arg}}$
LOG	1	natural logarithm: ln(arg)
LOG10	1	common logarithm: $\log_{10}$(arg)
EXP	1	exponential: exp(arg)
ABS	1	absolute (positive) value: |arg|
NINT	1	nearest integer value to arg
MAX	2 or more	maximum of arguments
MIN	2 or more	minimum of arguments
DIM	2	Fortran DIM (positive difference) function
MOD	2	Fortran MOD (remainder) function
SIGN	2	Fortran SIGN (transfer of sign) function
SIN*	1	sine function: sin(arg)
COS*	1	cosine function: cos(arg)
TAN*	1	tangent function: tan(arg)
ASIN*	1	inverse sine function: $\sin^{-1}$(arg)
ACOS*	1	inverse cosine function: $\cos^{-1}$(arg)
ATAN*	1	inverse tangent function: $\tan^{-1}$(arg)
ATAN2*	2	inverse tangent function: $\tan^{-1}$(arg1/arg2)
SINH*	1	hyperbolic sine function: sinh(arg)
COSH*	1	hyperbolic cosine function: cosh(arg)
TANH*	1	hyperbolic tangent function: tanh(arg)
SIND*	1	sine function: sin(arg)
COSD*	1	cosine function: cos(arg)
TAND*	1	tangent function: tan(arg)
ASIND*	1	inverse sine function: $\sin^{-1}$(arg)
ACOSD*	1	inverse cosine function: $\cos^{-1}$(arg)
ATAND*	1	inverse tangent function: $\tan^{-1}$(arg)
ATAN2D*	2	inverse tangent function: $\tan^{-1}$(arg1/arg2)

*Function does not support *integer* arithmetic.

The intrinsic functions recognised by **calc** (adapted from SUN/61). The angular arguments/results of the trigonometric functions are in radians, except those suffixed with a D, which are in degrees.

11 String Processing

The C-shell has no string-manipulation tools. Instead we mostly use the **echo** command and **awk** utility. The latter has its own language for text processing and you can write **awk** scripts to perform complex processing, normally from files. There are even books devoted to **awk** such as the succinctly titled *sed & awk* by Dale Dougherty (O'Reilly & Associates, 1990). Naturally enough a detailed description is far beyond the scope of this cookbook.

Other relevant tools include **cut**, **paste**, **grep** and **sed**. The last two and **awk** gain much of their power from regular expressions. A regular expression is a pattern of characters used to match the same characters in a search through text. The pattern can include special characters to refine the search. They include ones to anchor to the beginning or end of a line, select a specific number of characters, specify any characters and so on. If you are going to write lots of scripts which manipulate text, learning about regular expressions is time well spent.

Here we present a few one-line recipes. They can be included inline, but for clarity the values derived via **awk** are assigned to variables using the **set** command. Note that these may not be the most concise way of achieving the desired results.

11.1 String concatenation

To concatenate strings, you merely abut them.

```
set catal = "NGC "
set number = 2345
set object = "Processing $catal$number."
```

So `object` is assigned to `"Processing NGC 2345."`. Note that spaces must be enclosed in quotes. If you want to embed variable substitutions, these should be double quotes as above.

On occasions the variable name is ambiguous. Then you have to break up the string. Suppose you want `text` to be `"File cde1 is not abc"`. You can either make two abutted strings or encase the variable name in braces (`{ }`), as shown below.

```
set name1 = abc
set name = cde
set text = "File $name""1 is not $name1"     # These two assignments
set text = "File ${name}1 is not $name1"     # are equivalent.
```

Here are some other examples of string concatenation.

```
echo 'CIRCLE ( '$centre[1]', '$centre[2]', 20 )' > $file1".ard"
gausmooth in=$root"$suffix" out=${root}_sm accept
linplot ../arc/near_arc"($lbnd":"$ubnd)"
```

11.2 Obtain the length of a string

This requires either the **wc** command in an expression (see Section 10), or the **awk** function **length**. Here we determine the number of characters in variable `object` using both recipes.

```
set object = "Processing NGC 2345"
set nchar = `echo $object | awk '{print length($0)}'`      # = 19
@ nchar = `echo $object | wc -c` - 1
```

If the variable is an array, you can either obtain the length of the whole array or just an element. For the whole the number of characters is the length of a space-separated list of the elements. The double quotes are delimiters; they are not part of the string so are not counted.

```
set places = ( Jupiter "Eagle Nebula" "Gamma quadrant" )
set nchara = `echo $places | awk '{print length($0)}'`     # = 35
set nchar1 = `echo $places[1] | awk '{print length($0)}'`  # =  7
set nchar2 = `echo $places[2] | awk '{print length($0)}'`  # = 12
```

11.3 Find the position of a substring

This requires the **awk** function **index**. This returns zero if the string could not be located. Note that comparisons are case sensitive.

```
set catal = "NGC "
set number = 2345
set object = "Processing $catal$number."
set cind = `echo $object | awk '{print index($0,"ngc")}'`      # =  0
set cind = `echo $object | awk '{print index($0,"NGC")}'`      # = 12

set places = ( Jupiter "Eagle Nebula" "Gamma quadrant" )
set cposa = `echo $places | awk '{print index($0,"ebu")}'`     # = 16
set cposn = `echo $places | awk '{print index($0,"alpha")}'`   # =  0
set cpos1 = `echo $places[1] | awk '{print index($0,"Owl")}'`  # =  0
set cpos2 = `echo $places[2] | awk '{print index($0,"ebu")}'`  # =  8
set cpos3 = `echo $places[3] | awk '{print index($0,"rant")}'` # = 11
```

An array of strings is treated as a space-separated list of the elements. The double quotes are delimiters; they are not part of the string so are not counted.

11.4 Extracting a substring

One method uses the **awk** function **substr**(s,c,n). This returns the substring from string s starting from character position c up to a maximum length of n characters. If n is not supplied, the rest of the string from c is returned. Let's see it in action.

```
set caption = "Processing NGC 2345."
set object = `echo $caption | awk '{print substr($0,12,8)}'` # = "NGC 2345"
set objec_ = `echo $caption | awk '{print substr($0,16)}'`   # = "2345."

set places = ( Jupiter "Eagle Nebula" "Gamma quadrant" )
set oba = `echo $places | awk '{print substr($0,28,4)}'` # = "quad"
set ob1 = `echo $places[3] | awk '{print substr($0,7)}'` # = "quadrant"
```

An array of strings is treated as a space-separated list of the elements. The double quotes are delimiters; they are not part of the string so are not counted.

Another method uses the UNIX **cut** command. It too can specify a range or ranges of characters. It can also extract fields separated by nominated characters. Here are some examples using the same values for the array `places`

```
set cut1 = `echo $places | cut -d ' ' -f1,3`  # = "Jupiter Nebula"
set cut2 = `echo $places[3] | cut -d a -f2`   # = "mm"
set cut3 = `echo $places | cut -c3,11`        # = "pg"
set cut4 = `echo $places | cut -c3-11`        # = "piter Eag"
set cut5 = `cut -d ' ' -f1,3-5 table.dat`     # Extracts fields 1,3,4,5
                                              # from file table.dat
```

The `-d` qualifier specifies the delimiter between associated data (otherwise called fields). Note the the space delimiter must be quoted. The `-f` qualifier selects the fields. You can also select character columns with the `-c` qualifier. Both `-c` and `-f` can comprise a comma-separated list of individual values and/or ranges of values separated by a hyphen. As you might expect, **cut** can take its input from files too.

11.5 Split a string into an array

The **awk** function **split**(s,a,sep) splits a string s into an **awk** array a using the delimiter sep.

```
set time = 12:34:56
set hr = `echo $time | awk '{split($0,a,":"); print a[1]}'` # = 12
set sec = `echo $time | awk '{split($0,a,":"); print a[3]}'` # = 56

# = 12 34 56
set hms = `echo $time | awk '{split($0,a,":"); print a[1], a[2], a[3]}'`
set hms = `echo $time | awk '{split($0,a,":"); for (i=1; i<=3; i++) print a[i]}'`
```

The last two statements are equivalent, but the latter is more convenient for longer arrays as you specify the start index and number of elements to print. Variable `hms` is also an array so `hms[2]` is 34.

11.6 Changing case

Some implementations of **awk** offer functions to change case.

```
set text = "Eta-Aquarid shower"
set utext = `echo $text | awk '{print toupper($0)}'` # = "ETA-AQUARID SHOWER"
set ltext = `echo $text | awk '{print tolower($0)}'` # = "eta-aquarid shower"
```

11.7 String substitution

Some implementations of **awk** offer substitution functions **gsub**(e,s) and **sub**(e,s). The latter substitutes the s for the first match with the regular expression e in our supplied text. The former replaces every occurrence.

```
set text = "Eta-Aquarid shower"
# = "Eta-Aquarid stream"
set text = `echo $text | awk '{sub("shower","stream"); print $0}'`

# = "Eta-Aquxid strex"
set text1 = `echo $text | awk '{gsub("a[a-z]","x"); print $0}'`

# = "Eta-Aquaritt stream"
set text2 = `echo $text | awk '{sub("a*d","tt"); print $0}'`

set name = "Abell 3158"
set catalogue = `echo $name | awk '{sub("[0-9]+",""); print $0}'`  # = Abell
set short = `echo $name | awk '{gsub("[b-z]",""); print $0}'`      # = "A 3158"
```

There is also **sed**.

```
set text = `echo $text | sed 's/shower/stream/'`
```

is equivalent to the first **awk** example above. Similarly you could replace all occurrences.

```
set text1 = `echo $text | sed 's/a[a-z]/x/g'`
```

is equivalent to the second example. The final `g` requests that the substitution is applied to all occurrences.

12 Dealing with Files

12.1 Extracting parts of filenames

Occasionally you'll want to work with parts of a filename, such as the path or the file type. The C-shell provides filename modifiers that select the various portions. A couple are shown in the example below.

```
set type = $1:e
set name = $1:r
if ( $type == "bdf" ) then
   echo "Use BDF2NDF on a VAX to convert the Interim file $1"
else if ( $type != "dst" ) then
   hdstrace $name
else
   hdstrace @'"$1"'
endif
```

Suppose the first argument of a script, `$1`, is a filename called galaxy.bdf. The value of variable `type` is `bdf` and `name` equates to `galaxy` because of the presence of the filename modifiers `:e` and `:r`. The rest of the script uses the file type to control the processing, in this case to provide a listing of the contents of a data file using the HDSTRACE utility.

The complete list of modifiers, their meanings, and examples is presented in the table below.

Modifier	Value returned	Value for filename `/star/bin/kappa/comwest.sdf`
:e	Portion of the filename following a fullstop; if the filename does not contain a fullstop, it returns a null string	`sdf`
:r	Portion of the filename preceding a fullstop; if there is no fullstop present, it returns the complete filename	`comwest`
:h	The path of the filename (mnemonic: h for head)	`/star/bin/kappa`
:t	The tail of the file specification, excluding the path	`comwest.sdf`

12.2 Process a Series of Files

One of the most common things you'll want to do, having devised a data-processing path, is to apply those operations to a series of data files. For this you need a **foreach...end** construct.

```
convert              # Only need be invoked once per process
foreach file (*.fit)
   stats $file
end
```

This takes all the FITS files in the current directory and computes the statistics for them using the **stats** command from KAPPA. `file` is a shell variable. Each time in the loop `file` is assigned to the name of the next file of the list between the parentheses. The `*` is the familiar wildcard which matches any string. Remember when you want to use the shell variable's value you prefix it with a `$`. Thus `$file` is the filename.

12.2.1 NDFs

Some data formats like the NDF demand that only the file name (*i.e.* what appears before the last dot) be given in commands. To achieve this you must first strip off the remainder (the file extension or type) with the `:r` file modifier.

```
foreach file (*.sdf)
   histogram $file:r accept
end
```

This processes all the HDS files in the current directory and calculates an histogram for each of them using the **histogram** command from KAPPA. It assumes that the files are NDFs. The `:r` instructs the shell to remove the file extension (the part of the name following the the rightmost fullstop). If we didn't do this, the **histogram** task would try to find NDFs called SDF within each of the HDS files.

12.2.2 Wildcarded lists of files

You can give a list of files separated by spaces, each of which can include the various UNIX wildcards. Thus the code below would report the name of each NDF and its standard deviation. The NDFs are called 'Z' followed by a single character, ccd1, ccd2, ccd3, and spot.

```
foreach file (Z?.sdf ccd[1-3].sdf spot.sdf)
   echo "NDF:" $file:r"; sigma: "`stats $file:r | grep "Standard deviation"`
end
```

echo writes to standard output, so you can write text including values of shell variables to the screen or redirect it to a file. Thus the output produced by **stats** is piped (the `|` is the pipe) into the UNIX **grep** utility to search for the string `"Standard deviation"`. The `` ` ` `` invokes the command, and the resulting standard deviation is substituted.

You might just want to provide an arbitrary list of NDFs as arguments to a generic script. Suppose you had a script called `splotem`, and you have made it executable with `chmod +x splotem`.

```
#!/bin/csh
figaro                  # Only need be invoked once per process
foreach file ($*)
   if (-e $file) then
      splot $file:r accept
   endif
end
```

Notice the `-e` file-comparison operator. It tests whether the file exists or not. (Section 12.4 has a full list of the file operators.) To plot a series of spectra stored in NDFs, you just invoke it something like this.

```
% ./splotem myndf.sdf arc[a-z].sdf hd[0-9]*.sdf
```

See the glossary for a list of the available wildcards such as the `[a-z]` in the above example.

12.2.3 Exclude the .sdf for NDFs

In the `splotem` example from the previous section the list of NDFs on the command line required the inclusion of the `.sdf` file extension. Having to supply the `.sdf` for an NDF is abnormal. For reasons of familiarity and ease of use, you probably want your relevant scripts to accept a list of NDF names and to append the file extension automatically before the list is passed to **foreach**. So let's modify the previous example to do this.

```
#!/bin/csh
figaro                  # Only need be invoked once per process

#  Append the HDS file extension to the supplied arguments.
set ndfs
set i = 1
while ( $i <= $#argv )
   set ndfs = ($ndfs[*] $argv[i]".sdf")
   @ i = $i + 1
end

#  Plot each 1-dimensional NDFs.
foreach file ($ndfs[*])
   if (-e $file) then
      splot $file:r accept
   endif
end
```

This loops through all the arguments and appends the HDS-file extension to them by using a work array `ndfs`. The **set** defines a value for a shell variable; don't forget the spaces around the =. `ndfs[*]` means all the elements of variable `ndfs`. The loop adds elements to `ndfs` which is initialised without a value. Note the necessary parentheses around the expression `($ndfs[*] $argv[i]".sdf")`.

On the command line the wildcards have to be passed verbatim, because the shell will try to match with files than don't have the `.sdf` file extension. Thus you must protect the wildcards with quotes. It's a nuisance, but the advantages of wildcards more than compensate.

```
% ./splotem myndf 'arc[a-z]' 'hd[0-9]*'
% ./noise myndf 'ccd[a-z]'
```

If you forget to write the `' '`, you'll receive a `No match` error.

12.2.4 Examine a series of NDFs

A common need is to browse through several datasets, perhaps to locate a specific one, or to determine which are acceptable for further processing. The following presents images of a series of NDFs using the **display** task of KAPPA. The title of each plot tells you which NDF is currently displayed.

```
foreach file (*.sdf)
   display $file:r axes pltitl=$file:r
   sleep 5
end
```

sleep pauses the process for a given number of seconds, allowing you to view each image. If this is too inflexible you could add a prompt so the script displays the image once you press the return key.

```
set nfiles = `ls *.sdf | wc -w`
set i = 1
foreach file (*.sdf)
   display $file:r axes pltitl=$file:r

# Prompt for the next file unless it is the last.
   if ( $i < $nfiles ) then
      echo -n "Next?"
      set next = $<

# Increment the file counter by one.
      @ i++
   endif
end
```

The first lines shows a quick way of counting the number of files. It uses **ls** to expand the wildcards, then the command **wc** to count the number of words. The back quotes cause the instruction between them to be run and the values generated to be assigned to variable `nfiles`.

You can substitute another visualisation command for **display** as appropriate. You can also use the graphics database to plot more than one image on the screen or to hardcopy. The script `$KAPPA_DIR/multiplot.csh` does the latter.

12.3 Filename modification

Thus far the examples have not created a new file. When you want to create an output file, you need a name for it. This could be an explicit name, one derived from the process identification number, one generated by some counter, or from the input filename. Here we deal with all but the trivial first case.

12.3.1 Appending to the input filename

To help identify datasets and to indicate the processing steps used to generate them, their names are often created by appending suffices to the original file name. This is illustrated below.

```
foreach file (*.sdf)
   set ndf = $file:r
   block in=$ndf out=$ndf"_sm" accept
end
```

This uses **block** from KAPPA to perform block smoothing on a series of NDFs, creating new NDFs, each of which takes the name of the corresponding input NDF with a `_sm` suffix. The **accept** keyword accepts the suggested defaults for parameters that would otherwise be prompted. We use the **set** to assign the NDF name to variable `ndf` for clarity.

12.3.2 Appending a counter to the input filename

If a counter is preferred, this example

```
set count = 1
foreach file (*.sdf)
   set ndf = $file:r
   @ count = $count + 1
   block in=$ndf out=smooth$count accept
end
```

would behave as the previous one except that the output NDFs would be called `smooth1`, `smooth2` and so on.

12.3.3 Appending to the input filename

Whilst appending a suffix after each data-processing stage is feasible, it can generate some long names, which are tedious to handle. Instead you might want to replace part of the input name with a new string. The following creates another shell variable, `ndfout` by replacing the string `_flat` from the input NDF name with `_sm`. The script pipes the input name into the **sed** editor which performs the substitution.

```
foreach file (*_flat.sdf)
   set ndf = $file:r
   set ndfout = `echo $ndf | sed 's#_flat#_sm#'`
   block in=$ndf out=$ndfout accept
end
```

The `#` is a delimiter for the strings being substituted; it should be a character that is not present in the strings being altered. Notice the ` ` quotes in the assignment of `ndfout`. These instruct the shell to process the expression immediately, rather than treating it as a literal string. This is how you can put values output from UNIX commands and other applications into shell variables.

12.4 File operators

There is a special class of C-shell operator that lets you test the properties of a file. A file operator is used in comparison expressions of the form `if (file_operator file) then`. A list of file operators is tabulated to the right.

File operators	
Operator	**True if:**
`-d`	file is a directory
`-e`	file exists
`-f`	file is ordinary
`-o`	you are the owner of the file
`-r`	file is readable by you
`-w`	file is writable by you
`-x`	file is executable by you
`-z`	file is empty

The most common usage is to test for a file's existence. The following only runs **cleanup** if the first argument is an existing file.

```
# Check that the file given by the first
# argument exists before attempting to
# use it.
if ( -e $1 ) then
   cleanup $1
endif
```

Here are some other examples.

```
# Remove any empty directories.
if ( -d $file && -z $file ) then
   rmdir $file

# Give execute access to text files with a .csh extension.
else if ( $file:e == "csh" && -f $file ) then
   chmod +x $file
endif
```

12.5 Creating text files

A frequent feature of scripts is redirecting the output from tasks to a text file. For instance,

```
hdstrace $file:r > $file:r.lis
fitshead $fits > $$.tmp
```

directs the output of the **hdstrace** and **fitshead** to text files. The name of the first is generated from the name of the file whose contents are being listed, so for HDS file `cosmic.sdf` the trace is stored in `cosmic.lis`. In the second case, the process identification number is the name of the text file. You can include this special variable to generate the names of temporary files. (The `:r` is described in Section 12.1.)

If you intend to write more than once to a file you should first create the file with the **touch** command, and then append output to the file.

```
touch logfile.lis
foreach file (*.sdf)
   echo "FITS headers for $file:r:"  >> logfile.lis
   fitslist $file:r >> logfile.lis
   echo " "
end
```

Here we list FITS headers from a series of NDFs to file `logfile.lis`. There is a heading including the dataset name and blank line between each set of headers. Notice this time we use `>>` to append. If you try to redirect with `>` to an existing file you'll receive an error message whenever you have the `noclobber` variable set. `>!` redirects regardless of `noclobber`.

There is an alternative—write the text file as part of the script. This is often a better way for longer files. It utilises the **cat** command to create the file.

```
cat >! catpair_par.lis   <<EOF
${refndf}.txt
${compndf}.TXT
${compndf}_match.TXT
C
XPOS
YPOS
XPOS
YPOS
$distance
`echo $time | awk '{print substr($0,1,5)}'`
EOF
```

The above writes the text between the two `EOF`s to file `catpair_par.lis`. Note the second `EOF` must begin in column 1. You can choose the delimiting words; common ones are `EOF`, `FOO`. Remember the `>!` demands that the file be written regardless of whether the file already exists.

A handy feature is that you can embed shell variables, such as `refndf` and `distance` in the example. You can also include commands between left quotes (` `); the commands are evaluated before the file is written. However, should you want the special characters `$`, `\`, and ` ` to be treated literally insert a `\` before the delimiting word or a `\` before each special character.

12.5.1 Writing a script within a script

The last technique might be needed if your script writes another script, say for overnight batch processing, in which you want a command to be evaluated when the second script is run, not the first. You can also write files within this second script, provided you choose different words to delimit the file contents. Here's an example which combines both of these techniques.

```
cat >! /tmp/${user}_batch_${runno}.csh    <<EOF
#!/bin/csh

# Create the data file.
cat >! /tmp/${user}_data_${runno}  <<EOD
$runno
\`date\`
`star/bin/kappa/calc exp="LOG10($C+0.01*$runno)"`
EOD

<commands to perform the data processing using the data file>

# Remove the temporary script and data files.
rm /tmp/${user}batch_${runno}.csh
rm /tmp/${user}_data_${runno}

exit
EOF
chmod +x  /tmp/${user}_batch_${runno}.csh
```

This excerpt writes a script in the temporary directory. The temporary script's filename includes our username (`$user`) and some run number stored in variable `runno`. The temporary script begins with the standard comment indicating that it's a C-shell script. The script's first action is to write a three-line data file. Note the different delimiter, `EOD`. This data file is created only when the temporary script is run. As we want the time and date information at run time to be written to the data file, the command substitution backquotes are both escaped with a \. In contrast, the final line of the data file is evaluated before the temporary script is written. Finally, the temporary script removes itself and the data file. After making a temporary script, don't forget to give it execute permission.

12.6 Reading from text files

You can also read from a text file from within your script.

```
./doubleword < mynovel.txt

myprog <<BAR
320 512
$nstars
`wc -l < brightnesses.txt`
testimage
BAR

commentary <<\foo
   The AITCH package offers unrivalled facilities.
   It is also easy to use because of its GUI interface.

              Save $$$ if you buy now.
foo
```

Command `./doubleword` reads its standard input from the file `mynovel.txt`. The `<<`*word* obtains the input data from the script file itself until there is line beginning *word*. You may also include variables and commands to execute as the `$`, `\`, and `` ` ` `` retain their special meaning. If you want these characters to be treated literally, say to prevent substitution, insert a `\` before the delimiting *word*. The command `myprog` reads from the script, substituting the value of variable `nstars` in the second line, and the number of lines in file `brightnesses.txt` in the third line.

The technical term for such files are *here documents*.

12.7 Discarding text output

The output from some routines is often unwanted in scripts. In these cases redirect the standard output to a null file.

```
correlate in1=frame1 in2=frame2 out=framec > /dev/null
```

Here the text output from the task **correlate** is disposed of to the `/dev/null` file. Messages from Starlink tasks and usually Fortran channel 6 write to standard output.

12.8 Obtaining dataset attributes

When writing a data-processing pipeline connecting several applications you will often need to know some attribute of the data file, such as its number of dimensions, its shape, whether or not it may contain bad pixels, a variance array or a specified extension. The way to access these data is with **ndftrace** from KAPPA and **parget** commands. **ndftrace** inquires the data, and **parget** communicates the information to a shell variable.

12.8.1 Obtaining dataset shape

Suppose that you want to process all the two-dimensional NDFs in a directory. You would write something like this in your script.

```
foreach file (*.sdf)
   ndftrace $file:r > /dev/null
   set nodims = `parget ndim ndftrace`
   if ( $nodims == 2 ) then
      <perform the processing of the two-dimensional datasets>
   endif
end
```

Note although called **ndftrace**, this function can determine the properties of foreign data formats through the automatic conversion system (SUN/55, SSN/20). Of course, other formats do not have all the facilities of an NDF.

If you want the dimensions of a FITS file supplied as the first argument you need this ingredient.

```
ndftrace $1 > /dev/null
set dims = `parget dims ndftrace`
```

Then `dims[`i`]` will contain the size of the i^{th} dimension. Similarly

```
ndftrace $1 > /dev/null
set lbnd = `parget lbound ndftrace`
set ubnd = `parget ubound`
```

will assign the pixel bounds to arrays `lbnd` and `ubnd`.

12.8.2 Available attributes

Below is a complete list of the results parameters from **ndftrace**. If the parameter is an array, it will have one element per dimension of the data array (given by parameter NDIM); except for EXTNAM and EXTTYPE where there is one element per extension (given by parameter NEXTN). Several of the axis parameters are only set if the **ndftrace** input keyword `fullaxis` is set (not the default). To obtain, say, the data type of the axis centres of the current dataset, the code would look like this.

```
ndftrace fullaxis accept > dev/null
set axtype = `parget atype ndftrace`
```

Name	Array?	Meaning
AEND	Yes	The axis upper extents of the NDF. For non-monotonic axes, zero is used. See parameter AMONO. This is not assigned if AXIS is FALSE.
AFORM	Yes	The storage forms of the axis centres of the NDF. This is only written when parameter FULLAXIS is TRUE and AXIS is TRUE.
ALABEL	Yes	The axis labels of the NDF. This is not assigned if AXIS is FALSE.
AMONO	Yes	These are TRUE when the axis centres are monotonic, and FALSE otherwise. This is not assigned if AXIS is FALSE.
ANORM	Yes	The axis normalisation flags of the NDF. This is only written when FULLAXIS is TRUE and AXIS is TRUE.
ASTART	Yes	The axis lower extents of the NDF. For non-monotonic axes, zero is used. See parameter AMONO. This is not assigned if AXIS is FALSE.
ATYPE	Yes	The data types of the axis centres of the NDF. This is only written when FULLAXIS is TRUE and AXIS is TRUE.
AUNITS	Yes	The axis units of the NDF. This is not assigned if AXIS is FALSE.
AVARIANCE	Yes	Whether or not there are axis variance arrays present in the NDF. This is only written when FULLAXIS is TRUE and AXIS is TRUE.
AXIS		Whether or not the NDF has an axis system.
BAD		If TRUE, the NDF's data array may contain bad values.
BADBITS		The BADBITS mask. This is only valid when QUALITY is TRUE.
DIMS	Yes	The dimensions of the NDF.
EXTNAME	Yes	The names of the extensions in the NDF. It is only written when NEXTN is positive.
EXTTYPE	Yes	The types of the extensions in the NDF. Their order corresponds to the names in EXTNAME. It is only written when NEXTN is positive.
FORM		The storage form of the NDF's data array.
HISTORY		Whether or not the NDF contains HISTORY records.
LABEL		The label of the NDF.
LBOUND	Yes	The lower bounds of the NDF.
NDIM		The number of dimensions of the NDF.
NEXTN		The number of extensions in the NDF.
QUALITY		Whether or not the NDF contains a QUALITY array.
TITLE		The title of the NDF.
TYPE		The data type of the NDF's data array.
UBOUND	Yes	The upper bounds of the NDF.
UNITS		The units of the NDF.
VARIANCE		Whether or not the NDF contains a VARIANCE array.
WIDTH	Yes	Whether or not there are axis width arrays present in the NDF. This is only written when FULLAXIS is TRUE and AXIS is TRUE.

12.8.3 Does the dataset have variance/quality/axis/history information?

Suppose you have an application which demands that variance information be present, say for optimal extraction of spectra, you could test for the existence of a variance array in your FITS file called `dataset.fit` like this.

```
#  Enable automatic conversion
convert          # Needs to be invoked only once per process

set file = dataset.fit
ndftrace $file > /dev/null
set varpres = `parget variance ndftrace`
if ( $varpres == "FALSE" ) then
   echo "File $file does not contain variance information"
else
   <process the dataset>
endif
```

The logical results parameters have values `TRUE` or `FALSE`. You merely substitute another component such as quality or axis in the **parget** command to test for the presence of these components.

12.8.4 Testing for bad pixels

Imagine you have an application which could not process bad pixels. You could test whether a dataset *might* contain bad pixels, and run some pre-processing task to remove them first. This attribute could be inquired via **ndftrace**. If you need to know whether or not any were actually present, you should run **setbad** from KAPPA first.

```
setbad $file
ndftrace $file > /dev/null
set badpix = `parget bad ndftrace`
if ( badpix == "TRUE" ) then
   <remove the bad pixels>
else
   goto tidy
endif
<perform data processing>

tidy:
<tidy any temporary files, windows etc.>
exit
```

Here we also introduce the **goto** command—yes there really is one. It is usually reserved for exiting (`goto exit`), or, as here, moving to a named label. This lets us skip over some code, and move directly to the closedown tidying operations. Notice the colon terminating the label itself, and that it is absent from the **goto** command.

12.8.5 Testing for a spectral dataset

One recipe for testing for a spectrum is to look at the axis labels. Here is a longer example showing how this might be implemented. Suppose the name of the dataset being probed is stored in variable `ndf`.

```
# Get the full attributes.
ndftrace $ndf fullaxis accept > /dev/null

# Assign the axis labels and number of dimensions to variables.
set axlabel = `parget atype ndftrace`
set nodims = `parget ndim`

# Exit the script when there are too many dimensions to handle.
if ( $nodims > 2 ) then
   echo Cannot process a $nodims-dimensional dataset.
   goto exit
endif

# Loop for each dimension or until a spectral axis is detected.
set i = 1
set spectrum = FALSE
while ( $i <= nodims && $spectrum == FALSE )

# For simplicity the definition of a spectral axis is that
# the axis label is one of a list of acceptable values.  This
# test could be made more sophisticated.  The toupper converts the
# label to uppercase to simplify the comparison.  Note the \ line
# continuation.
   set uaxlabel = `echo $axlabel[$i] | awk '{print toupper($0)}'`
   if ( $uaxlabel == "WAVELENGTH" || \
        $uaxlabel == "VELOCITY" ) then

# Record that the axis is found and which dimension it is.
      set spectrum = TRUE
      set spaxis = $i
   endif
   @ i++
end

# Process the spectrum.
if ( $spectrum == TRUE ) then

# Rotate the dataset to make the spectral axis along the first
# dimension.
   if ( $spaxis == 2 ) then
      irot90 $file $file"_rot" accept

# Fit the continuum.
      sfit spectrum=$file"_rot" order=2 output=$file"_fit" accept
   else
      sfit spectrum=$file order=2 output=$file"_fit accept
   end if
endif
```

12.9 FITS Headers

Associated with FITS files and many NDFs is header information stored in 80-character 'cards'. It is possible to use these ancillary data in your script. Each non-comment header has a keyword, by which you can reference it; a value; and usually a comment. KAPPA from V0.10 has a few commands for processing FITS header information described in the following sections.

12.9.1 Testing for the existence of a FITS header value

Suppose that you wanted to determine whether an NDF called image123 contains an AIRMASS keyword in its FITS headers (stored in the FITS extension).

```
set airpres = `fitsexist image123 airmass`
if ( $airpres == "TRUE" ) then
   <access AIRMASS FITS header>
endif
```

Variable `airpres` would be assigned `"TRUE"` when the AIRMASS card was present, and `"FALSE"` otherwise. Remember that the ` ` quotes cause the enclosed command to be executed.

12.9.2 Reading a FITS header value

Once we know the named header exists, we can then assign its value to a shell variable.

```
set airpres = `fitsexist image123 airmass`
if ( $airpres == "TRUE" ) then
   set airmass = `fitsval image123 airmass`
   echo "The airmass for image123 is $airmass."
endif
```

12.9.3 Writing or modifying a FITS header value

We can also write new headers at specified locations (the default being just before the END card), or revise the value and/or comment of existing headers. As we know the header AIRMASS exists in image123, the following revises the value and comment of the AIRMASS header. It also writes a new header called FILTER immediately preceding the AIRMASS card assigning it value `B` and comment `Waveband`.

```
fitswrite image123 airmass value=1.062 comment=\"Corrected airmass\"
fitswrite image123 filter position=airmass value=B comment=Waveband
```

As we want the metacharacters `"` to be treated literally, each is preceded by a backslash.

12.10 Accessing other objects

You can manipulate data objects in HDS files, such as components of an NDF's extension. There are several Starlink applications for this purpose including the FIGARO commands **copobj**, **creobj**, **delobj**, **renobj**, **setobj**; and the KAPPA commands **setext**, and **erase**.

For example, if you wanted to obtain the value of the EPOCH object from an extension called IRAS_ASTROMETRY in an NDF called lmc, you could do it like this.

```
set year = `setext lmc xname=iras_astrometry option=get \
            cname=epoch noloop`
```

The `noloop` prevents prompting for another extension-editing operation. The single backslash is the line continuation.

12.11 Defining NDF sections with variables

If you want to define a subset or superset of a dataset, most Starlink applications recognise NDF sections (see SUN/95's chapter called "NDF Sections") appended after the name.

A naïve approach might expect the following to work

```
set lbnd = 50
set ubnd = 120
linplot $KAPPA_DIR/spectrum"($lbnd:$ubnd)"
display $KAPPA_DIR/comwest"($lbnd:$ubnd",$lbnd~$ubnd)"
```

however, they generate the `Bad : modifier in $ ($).` error. That's because it is stupidly looking for a filename modifier `:$` (see Section 12.1).

Instead here are some recipes that work.

```
set lbnd = 50
set ubnd = 120
set lrange = "101:150"

linplot $KAPPA_DIR/spectrum"($lbnd":"$ubnd)"
stats abc"(-20:99,~$ubnd)"
display $KAPPA_DIR/comwest"($lbnd":"$ubnd",$lbnd":"$ubnd")"
histogram hale-bopp.fit'('$lbnd':'$ubnd','$lbnd':'$ubnd')'
ndfcopy $file1.imh"("$lbnd":"$ubnd","$lrange")" $work"1"
splot hd102456'('$ubnd~60')'
```

An easy-to-remember formula is to enclose the parentheses and colons in quotes.

13 Loop a specified number of times

Suppose that you want to loop a specified number of times. For this you need a **while** statement, as in this example.

```
set count = 1
set tempin = myndf
set box = 23
while ($count < 5)
   @ box = $box - $count * 2
   block in=$tempin out=tempndf box=$box
   mv tempndf.sdf myndf_sm4.sdf
   set tempin = myndf_sm4
   @ count = $count + 1
end
```

This performs four block smoothing operations, with a decreasing box size, on a NDF called myndf, the final result being stored in the NDF called myndf_sm4. The **while** statement tests a conditional statement and loops if it is true. So here it loops whenever the value of `count` is less than 5. For a list of the available operators see Section 10.1 or the **man** pages.

```
% man csh
/Expressions
```

The box size for the smoothing is evaluated in an expression `@ box = $box - $count * 2`. Note the space between the `@` and the variable, and the spaces around the `=`. Thus the box sizes will be 21, 17, 11, 3. Further details are in Section 10.1. You should also give the variable a value with **set** before assigning it an expression. Another expression increments the counter `count`. The NDF called tempndf is overwritten for each smooth using the standard UNIX command **mv**.

14 UNIX-style options

If you want other arguments, or a UNIX-like set of *options* for specifying arguments with defaults, you'll need to use **switch** and **shift** statements, rather than the simple **while** loop.

Suppose that we want to obtain photometric details of a galaxy, known to be the brightest object in a series of NDFs. Before that's possible we must first measure the background in each image. To reduce contamination that could bias the determination of the sky we mask the region containing the galaxy. Here we obtain the values for the shape, orientation, and the names of the NDFs, and assign them to shell variables. There are also default values.

```
#  Initialize some shell variables, including the default shape
#  and orientation.
set major = 82
set minor = 44
set orient = 152
set args = ($argv[1-])
set ndfs

#  Check that there are some arguments present.
if ( $#args == 0 ) then
   echo "Usage: galmask [-a semi-major] [-b semi-minor] " \
                       "[-o orientation] ndf1 [ndf2...]"
   exit
endif

#  Process each of the arguments to the script.
while ( $#args > 0 )
   switch ($args[1])
   case -a:        #  Semi-major axis
      shift args
      set major = $args[1]
      shift args
      breaksw
   case -b:        #  Semi-minor axis
      shift args
      set minor = $args[1]
      shift args
      breaksw
   case -o:        #  Orientation
      shift args
      set orient = $args[1]
      shift args
      breaksw
   case *:         #  The NDFs
      set ndfs = ($ndfs[1-] $args[1])
      shift args
      breaksw
   endsw
end

# Loop through the remaining arguments, assuming that these are NDFs.
foreach file ($ndfs[*])
   :      :      :
```

Some defaults are defined so that if the argument is not present, there is a suitable value. So for instance, the ellipse orientation is 152° unless there is a `-o` *orientation-value* on the command line when invoking the script.

The **switch** looks for specific arguments. If the first argument matches one of the allowed cases: `-a`, `-b`, `-o`, or any string (*) in that order, the script jumps to that case, and follows the commands there until it encounters the **breaksw**, and then the script moves to the **endsw** statement. You may be wondering why only the first element of the arguments is tested against the cases. This is where the **shift** statement comes in. This decrements the element indices of an array by one, so that `$argv[1]` is lost, `$argv[2]` becomes `$argv[1]`, `$argv[3]` becomes `$argv[2]` and so on. If we look at the `-a` case. The arguments are shifted, so that the first argument is now the value of the semi-major axis. This in turn is shifted out of the arguments. The * case does the equivalent of the earlier **while** loop (see Section 7.6) as it appends the NDF filenames. The `[1-]` means the first element until the last.

To actually perform the masking we could write an ARD ellipse shape using the variables, and pipe it into a file, in this case `biggal.ard`.

```
echo 'ELLIPSE( '$centre[1]', '$centre[2]', '$major', '$minor', '$orient' )' \
     > biggal.ard
```

On the command line we could enter

```
% galmask -o 145 -a 78.4 myndf 'ccd*o'
```

if the script was called `galmask`. This would set the semi-major axis to 78.4 pixels, the orientation to 145°, leaving the semi-minor axis at its default of 44 pixels. It would operate on the NDF called myndf and those whose names began with `ccd` and ended with `o`.

There is a related example in Section 17.1 where the ellipse axis lengths and orientation are fixed for two galaxies. The above options code could replace that recipe's first two lines, if we wanted to alter the spatial parameters for the brighter galaxy to see which gives best results.

In `$KAPPA_DIR/multiplot.csh` you can find a complete example. For a series of NDFs, this produces a grid of displayed images with axes and titles, prints them to a nominated device, once each page is full. You can modify it for other operations, where a single page of graphics output is produced by running several tasks, and joining them together with PSMERGE (SUN/164).

15 Debugging scripts

Earlier in Section 5 a number of ways to run a script were presented. There is yet another, which makes debugging easier. If you invoke the script with the **csh** command you can specify none or one or more options. The **command** takes the form

```
% csh [options] script
```

where `script` is the file. There is a choice of two debugging options.

```
% csh -x myscript
% csh -v myscript
```

The **x** option echoes the command lines after variable substitution, and the **v** option echoes the command lines before variable substitution. The **v** is normally used to identify the line at which the script is failing, and the **x** option helps to locate errors in variable substitution. As these options can produce lots of output flashing past your eyes, it is often sensible to redirect the output to a file to be examined at a comfortable pace.

If you prefer, you can run the script in the normal way by putting the option into the first comment line.

```
#!/bin/csh -x
```

16 Breaking-in

Sometimes you will realise you have made a mistake after starting a script, and therefore want to stop it to save time or damage. You can break-in using `CTRL/C` (hitting C while depressing the CTRL key), but doing this can leave unwanted intermediate files, graphics windows, and other mess. However, if you have

```
onintr label

<the main body of the script>

label:
<perform housekeeping>

exit
```

near the head and tail of your script, whenever `CTRL/C` is detected, the script will go to the specified label. There you can close down the script in a controlled fashion, removing the garbage. If you want to prevent `CTRL/C` interrupts, you should include the line

```
% onintr -
```

instead.

17 Longer recipes

This section brings together a number of the individual ingredients seen earlier into lengthier scripts. So far there is only one.

17.1 Recipe for masking and background-fitting

The following script fits a surface to the background in a series of NDFs. First two bright galaxies are to be excluded from the fitting process by being masked using an ARD region (see SUN/95, "Doing it the ARD way"). It is known that the brighter galaxy is the brightest object in the field and the relative displacement of the two galaxies is constant.

```
# Loop through the remaining arguments, assuming that these are NDFs.
foreach file ($ndfs[1-])

# Obtain the NDF's name.
   set file1=$file:r

# Obtain the centre of the galaxy, assuming it is the pixel with the
# largest value.
   stats $file1 > /dev/null

# Store the maximum (centre) co-ordinates values in shell variables.
   set centre = `parget maxcoord stats`

# Create a two-line ARD file.  The first is for the bright galaxy, and
# the other is the second brightest galaxy.  We assume a constant offset.
# Use CALC to evaluate the expressions, as the centres are strings and
# might be floating point.
   echo 'ELLIPSE( '$centre[1]', '$centre[2]', 82, 44, 152 )' > $file1".ard"

   set aa = `calc exp=\"$centre[1] + 68\"`
   set bb = `calc exp=\"$centre[2] - 59\"`

   echo 'ELLIPSE( '$aa', '$bb', 30, 25, 105 )' >> $file1".ard"

# Mask the NDF.
   ardmask $file1 $file1".ard" $file1"_masked" cosys=w
   rm $file1.ard

# Do the surface fit.
   echo " "
   echo $file:r
   surfit in=$file1"_masked" out=$file1"_bg" estimator=median \
          fittype=polynomial nxpar=4 nypar=4 ix=16 iy=16 \
          fitclip=\[2,2.5,3\] evaluate=interpolate

# Perform the sky subtraction.
   sub $file1 $file1"_bg" $file1"_ss"

# Remove work files.
    rm $file1"_bg.sdf" $file1"_masked.sdf"
```

```
end

exit
```

parget obtains the x-y co-ordinates of the maximum value in each NDF, and these are stored in variable `centre`. Since there are two values, `centre` is an array. The value of the first element `$centre[1]` is the x co-ordinate from **stats**, and the second `$centre[2]` is the y co-ordinate. These values are placed in an ARD expression and output to `$file".ard"`. **calc** applies the offset between the galaxies. The second ARD expression is appended to the same text file. **ardmask** then masks the galaxies, **surfit** performs the background fit, and subtracts it from the original NDF. The intermediate files are removed. The **exit** command meaning exit the script is implied at the end of the file, so it can usually be omitted. You might want to call it to leave the script, when some error is encountered.

If you want to make the shape and orientation arguments of the script, see the example of UNIX-style options in Section 14.

18 Glossary

- **Alias**
  A mechanism for abbreviating a C-shell command line.

- **ARD**
  ASCII Region Definition. A syntax for specifying various regions of an image in a text file. Used for masking image data. It is described in SUN/183.

- **awk**
  A pattern-scanning and text-processing language. In other words it is a programmable report generator. Its name comes from the initials of the authors.

- **CONVERT**
  A Starlink utility package for converting between NDF and various data formats such as FITS. It is described in SUN/55.

- **Current process**
  The task currently running the shell (in the context of this document). C-shell scripts invoked from the current process are each run in new (child) processes.

- **Environment variable**
  A global variable to define the characteristics of a UNIX environment such as the terminal type or home directory. It is defined with the **setenv** command. By convention, environment variables appear in uppercase. Environment variables are often used to refer to a directory.

- **Figaro**
  A general astronomical data-reduction package but concentrating on spectroscopy. It is available in several flavours. The Starlink version is described in SUN/86.

- **File modifier**
  Syntax for specifying a part of a filename. They appear following a filename and have the form colon followed by a letter. For example, `:t` excludes the path.

- **FITS**
  Flexible Image Transport System (FITS). The most commonly used format for astronomical data storage. It comprises a series of text headers followed by image or tabular data.

- **HDS**
  Hierarchical Data System. The underlying data system for Starlink data files. It is used to create and access NDF datasets.

- **KAPPA**
  The Starlink Kernel Application Package. A suite of facilities for processing and viewing astronomical images. Described in SUN/95.

- **Match**
  A string in a file that is successfully specified by a regular expression. It is also a filename, shell variable, or alias successfully specified by wildcards within the shell.

- **Metacharacter**
  Characters which have special meanings. For the shell these include wildcards, quotes, and logical operators. In regular expressions, metacharacters are used to specify strings to match.

- **NDF**
  The standard Starlink data-storage format. An hierarchical format for multi-dimensional data storage. Accessed using libraries supported by Starlink. Use of NDF is described in SUN/33.

- **NDF Section**
  A subset or superset of a dataset (originally applied to just NDFs) defined by specifying the pixel bounds or co-ordinate limits along each axis following the dataset's name. See SUN/95's chapter called "NDF Sections" for a description and many examples.

- **Output parameters**
  A channel through which some Starlink applications write their results. These tend to be applications that do not create an output dataset, such as statistics and dataset attributes. They are sometimes called "Results parameters".

- **Path**
  A list of directories which the system searches in turn to resolve command requests.

- **Pipe, piping**
  Mechanism by which the standard output of one programme is passed to the standard input of another. It allows sophisticated tools to be made from simple commands. The | character represents the pipe.

- **Process**
  A task being performed by the computer.

- **Process identification number**
  A positive integer that uniquely identifies a process within the system.

- **Regular expression**
  A pattern of characters used to match against the same characters in a search. They usually include special characters, which represent things other than themselves, to refine the search. Regular expressions empower utilities like **grep**, **sed** and **awk**. Although similar to shell wildcards there are differences so be careful.

- **sed**
  The stream editor. It is useful for editing large files or applying the same edits to a series of files.

- **Shell**
  A programme that listens to your terminal, and accepts and interprets the commands you type. There are several UNIX shells including the Bourne (sh), Bourne-again (bash), Korn (ksh), as well as the C shell (csh).

- **Shell variable**
  An identifier that can store one or more strings. Variables enable string processing, and integer and logical expressions in the shell.

- **Standard input**
  The file from which most UNIX programmes read their input data. It defaults to the terminal if you do not supply the input on the command line or a file.

- **Standard output**
  File to which most UNIX programmes output their results. Text output from Starlink applications are also routed there. Standard output defaults to your terminal. It can be piped into commands that accept standard input.

- **Starlink**
  UK network of computers for astronomers; a collection of software to reduce and analyse astronomical data; and the team of people supporting this hardware and software.

- **Wildcards**
  A shorthand notation to specify filenames, aliases or shell variables by supplying a certain special characters that represent things other than themselves.

Wildcard expression	Matches
`*`	Zero or more characters
`?`	Exactly one character
`[xyz]`	One character in the set `x`, `y`, or `z`
`[a-m]`	One character in the range from `a` to `m`
`[A-Za-z]`	All alphabetic characters
`[0-9]`	All numeric characters
`{alpha,beta,a,b}`	A set of options, `alpha`, `beta`, `a`, or `b`
`{aa,bb[1-3]}`	`aa`, `bb1`, `bb2`, or `bb3`